NPS-CS-05-008

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

FINAL REPORT ----- ACOUSTIC SIMULATION API REQUIREMENTS MODEL

by

Valdis Berzins

September 2005

Approved for public release; distribution is unlimited

Prepared for: NAVAIR
12350 Research Parkway, Orlando, FL

This report was prepared for and funded by NAVAIR.

Prepared by:

Valdis Berzins
Professor
Computer Science

Reviewed by:

Peter Denning
Chair
Computer Science

Released by:

Leonard A. Ferrari
Associate Provost and
Dean of Research

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** 7/27/2005	**3. REPORT TYPE AND DATES COVERED** Final Report FY05	
4. TITLE AND SUBTITLE: Title (Mix case letters) Acoustic Simulation API Requirements Model Final Report FY 05			**5. FUNDING NUMBERS** N6133905WX00302
6. AUTHOR(S) V. Berzins			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER** NPS-CS-05-008
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) NAVAIR 12350 Research Parkway, Orlando, FL			**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this report are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Distribution Statement (mix case letters)			**12b. DISTRIBUTION CODE**
13. ABSTRACT *(maximum 200 words)* This report describes a model-independent API for ocean acoustic simulation. It is intended primarily to support training applications and to decouple them from the detail of particular acoustic models			
14. SUBJECT TERMS Acoustic Simulation			**15. NUMBER OF PAGES** 41
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

Table of Contents

1 INTRODUCTION

1.1 Purpose

The purpose of the Acoustic Simulation API is to provide a standard interface to a variety of ocean acoustic models meeting the needs of application developers. The initially envisioned application domain is simulation support for training of sonar operators. Other likely applications include future mission planning, simulating artificial environments to support test and evaluation of new fleet systems, comparative scientific studies of different acoustic models and other unanticipated uses.

The interface was designed to support the most popular models, including the Enhanced EVA model, KingKong/FeyRay model, and the CASS/GRAB model. A long range goal is to accommodate other models as well with a minimum of incompatible modifications. The object-oriented design of the API was chosen to enable compatible extensions for this purpose by simply adding more classes and operations, ideally without changes to the interface classes already defined.

1.2 Context

Acoustic models are used for computer simulation of the propagation of sound through the ocean. This problem is not amenable to closed form mathematical solutions because the speed of sound in the ocean, and hence also the index of refraction for sound waves, is not uniform. The main factors influencing the speed of sound are the depth, salinity, and water temperature.

The direction of sound propagation can be affected by non-uniformities in the index of refraction as well as reflection off the surface and the bottom. Sound can be confined to a shallow, isothermal layer, called a surface duct, by surface reflections and refractions due to the upward refracting nature of the sound speed profile in the duct. A deeper local minimum in index of refraction can also act as a waveguide. Reflections can cause loss of signal strength, particularly reflections from the bottom or a rough surface. Sound in the ocean can follow more than one path between two points due to differences in launch angles and numbers of reflections. Different paths can have different signal strengths, different time delays, and different phase offsets that can cause interference effects.

Ray tracing models are commonly used in acoustic simulation because they are computationally efficient. All three of the legacy models considered in this study (Fey Ray, EVA, and CASS/GRAB) employ ray tracing. Some alternative classes of computational models for acoustic modeling include 1) normal mode models 2) parabolic equation models, and 3) wave number integration models. These types of models can provide more accuracy at the expense of more computation time. The API proposed here was designed without regard for these alternative types of acoustic models.

Several Oceanographic Environmental Acoustic Propagation Loss models have been developed to model acoustic propagation and reverberation in real-world ocean environments, including the three models we consider in this study. However, due to independent development, these models have variations in how they interface to applications and provide output data. Therefore, different high-level application codes are currently needed to invoke different models. This increases development costs of training systems, makes it difficult to compare the results of different models, and makes it difficult to port existing applications to new models. At the current time there are no accepted standards for interfaces to oceanographic acoustic models. The API proposed in this document is a first step toward development of such a standard.

1.3 Concept of Operation

Our goal is to develop an Application Program Interface that is common to the above-mentioned models and future models. This API provides a set of common services related to acoustic measurements to support training and simulation applications. The existing models are legacy systems that cannot be modified. Hence an additional layer of software wrappers (with relatively low execution overhead) that realize proposed API services in terms of legacy services is recommended. The purpose of the wrappers is to repackage legacy services to match a common standard API, not to implement new services. This restriction has limited the expressive power and convenience to developers of the current version of the API.

The diagram shown in Fig.1 summarizes our vision of the interaction between a typical training application, the proposed API, and a selected legacy acoustic model.

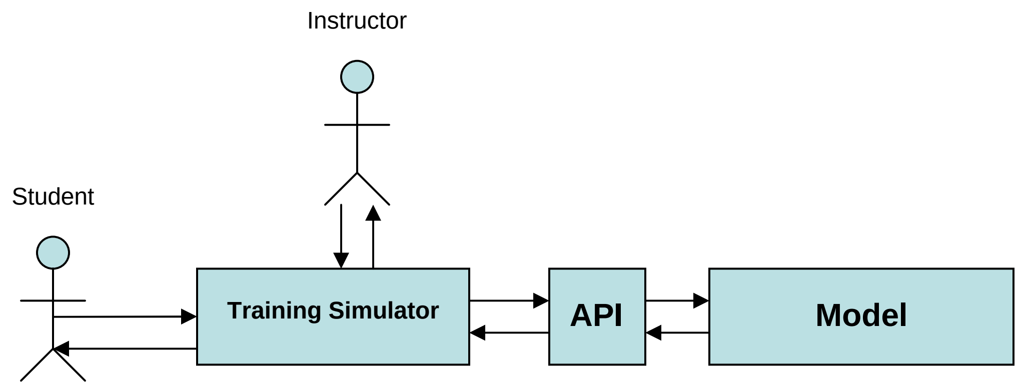


Figure 1- API Context Diagram

Implicit in the proposed concept of operation are the following divisions of responsibilities:

1. The API and the underlying acoustic model are responsible for:
    A. simulating passive sensor outputs
    B. simulating active sensor outputs
2. The applications using the API are responsible for:

A. any signal processing needed for determining higher level properties of sensor outputs, such as spectral analysis.

B. deriving and presenting various plots and displays to users, such as instructors and students, based on simulated sensor outputs.

C. interacting with users as necessary to determine necessary attributes of the simulation scenario, such as the gaming area, locations of active sources, reflectors, sensors and other relevant attributes.

D. interacting with users as necessary for authentication and obtaining any id's and passwords needed to access the acoustic models.

1.4 Organization of the API

The main API is structured as an abstract class with one concrete subclass for each legacy model, as shown in Fig. 2. The abstract class defines the set of services that are(or should be) common to all of the legacy models. The model-specific subclasses augment these with additional services and data models specific to each legacy model. The implications of this structure are:

1. The applications programmer must create an instance of the API class before using any of the services it provides.
2. Since the part of the API common to all models API is an abstract class, it cannot be instantiated directly. The applications programmer must instantiate one of the concrete subclasses instead, thereby choosing a specific legacy model to be used.
3. This structure can support concurrent use of instances of several *different* legacy models. Note however, that the current version of the design does not support concurrent use of more than one instance of the *same* legacy model.

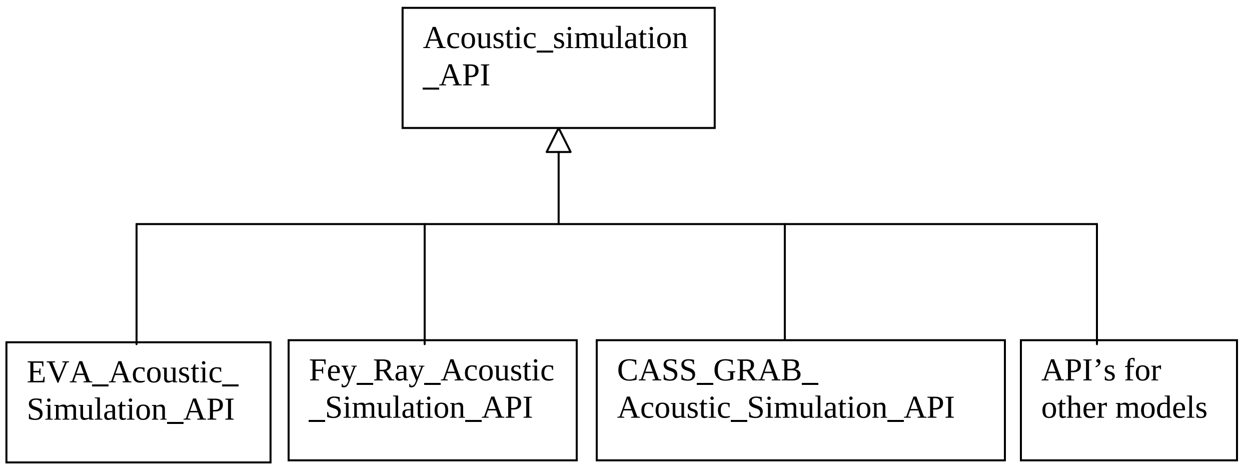

Acoustic_simulation_API
EVA_Acoustic_Simulation_API
Fey_Ray_Acoustic_Simulation_API
CASS_GRAB_Acoustic_Simulation_API
API's for other models

Fig. 2 API Class Structure

The API also contains a number of auxiliary classes that model input and output data used by the services provided by the main API. This enables the same interface to connect to several different models that use different data structures to represent the same information.

1.5 Summary of Essential Services

An essential model represents an ideal system in the absence of technology and performance constraints. The essential services of the acoustic simulation API are listed in Fig. 3.

Get_Sensor_Output

Discrete_Reverberation

Continuous_Reverberation

Propagation_loss_vs_range

Point_To_Point_Propagation_Loss

Fig. 3 List of essential API Services

1.6 Practical Issues and API Rationale

The practical API contains many more services than the ideal set listed in Fig. 3 because:

1. Input parameters of services have been transformed into persistent attributes that can be set and examined by a set of auxiliary input and output services.
2. Support services to handle issues such as system initialization have been added.
3. A set of optional controls for tuning performance and quality of service have been added.

These issues are discussed further in the rest of this section.

1.6.1 Optimizations

The use of persistent attributes to represent input parameters describing the simulation scenario is an optimization that reduces the number of times that (possibly voluminous) information needs to be transmitted (possibly over a network) and that expensive computation (such as reverberation) must be repeated. Small perturbations in positions of sources and targets can be accommodated by extrapolation based on derivatives (e.g in EVA), rather than by full recomputation if previous values of inputs and outputs are stored.

These optimizations are useful because many aspects of the simulation scenario, such as water properties and ocean bottom properties, change very slowly or remain constant over the course of a simulation session. In normal usage, the input attributes will be set by the application before the main services are required.

1.6.2 Initialization of Input Attributes

To ensure that the state of the API is fully observable to the application, all input attributes are subject to the following uniform requirement:

R1: All input attributes can be both set and retrieved via the API.

This frees the application from the need to keep track of the current values of the input attributes, and gives testers full observability into the state of the API. This motivates the convention explained in section 2.1: every attribute_setting operation has a matching attribute_value retrieval operation.

Persistent attributes require initialization, and could produce runtime errors if the main services are invoked before the input attributes on which they depend have been set. To prevent such errors, the API has been defined as a set of object classes whose instances provide all acoustic simulation services. To get access to the services provided by the API, the application must create an instance of the appropriate class by invoking a constructor operation. The constructor operations are responsible for the following issues:

1. Initialization of the underlying acoustic model to make it ready for operations. This can include actions such as setting up network connections, loading data from associated databases, obtaining passwords needed for access, allocating processors, memory, and other computing resources, and so on.
2. Initialization of all the input attributes on which other API services depend. The initial values must be either specified meaningful defaults or given by the applications as input parameters to the constructor operation.

Proper implementation of the API constructor operations will therefore prevent potential errors due to access to initially attribute.

The constructor operations also control the choice of which underlying acoustic model is used, and enable future applications to run instances of several different models concurrently, either for purposes of comparison or to provide different qualities of service (e.g., fast, rough results vs. slow, but more accurate ones).

1.6.3 Performance Tuning Controls

There are optional performance tuning services unique to each model. All of these are subject to the following uniform requirement:

R2: The model must function correctly and perform reasonably well even if the performance tuning services are never called.

This requirement implies that most users of the API will be able to ignore the existence of the performance tuning services. In an ideal situation, the effect of each performance tuning parameter should be clear to an application domain expert who is not familiar with the details of internal operation of the underlying model. This issue is an open problem at this point because many of the performance tuning operations provided by the legacy models are not explained from the point of view of a user (rather than that of a developer), and many of these operations are poorly documented. It is therefore likely that the typical user of the API will not be able to understand what the performance tuning operations do. In the current version of the API, application developers who need to tune performance have to do their own in-depth investigations of the effect of the performance tuning parameters on the specific model they are using.

In the current version of the API, all performance tuning control services are model-specific, and are located in the model-specific concrete subclass of the main API. Eventually it could be useful to create a standard, model-independent performance tuning interface. This will, however, require substantial further analysis and possible re-engineering of parts of the legacy models. This was not done in the current API because projected costs exceeded available resources.

2 OUTPUT SERVICES AND ATTRIBUTES

General policies:

- Every Set_x service has a corresponding Get_x service, for any persistent attribute x. These attribute output services are not explicitly listed here to avoid clutter.
- Every service can throw the exception "not_supported" if the current implementation of the specific model used does not support the service. This policy enables the API to provide some useful services even if some of the legacy models do not support them at the current time.

2.1 Implicitly Defined Attribute Access Functions

This section outlines the uniform conventions for packaging services related to persistent input attributes that apply to the proposed API.

If a class C has a settable attribute A of type T, it will have an operation for setting the attribute with the following interface.

```
Set_A (x: C, value: T)
```

The effect of this operation is to set attribute A of object x to the given value.
Each such operation has a companion readout operation with the following interface:

```
Get_A (x: C): T
```

The effect of the operation get_A is to return the current value of attribute A of object x.

Every set_A operation has a matching get_A operation, for each attribute A of each class C in the API. These operations are not explicitly specified to avoid cluttering document.

There are also some computed attributes that are not settable. These are defined explicitly in the document.

Game_Clock Gaming_Area

Fig. 4 List of Settable Attributes for the Acoustic simulation API

There are also a number of attributes needed to support the simulation that are not explicitly input by the user, but rather are drawn from databases internal to the legacy models. There are used internally by the models and are not exported by the current version of the API. For example, EVA has internally databases that supply the following kinds of information :

1. ocean depth
2. surface temperature
3. bottom loss characteristics
4. magnetic variation and inclination
5. current speed and direction at a requested depth
6. water temperature at a requested depth
7. sound velocity at a requested depth
8. salinity at a requested depth

2.2 Get_Sensor_Output

Service: get_sensor_output

EVA	- not provided
Fey Rey	- not provided
CASS/GRAB	- not in DRAB, CASS has services, detail not determined

Input:	Receiver: sensor
	Receiver_location: location,
	Receiver_orientation: orientation
	Game clock: integer
	//seconds since game start acoustic pressure level
Output:	amplitude: float

This operation was judged to be the most useful to the developer of training simulations. However, it is not directly supported by any of the legacy models, which provide lower level services. The operation was not modeled in more detail because of lack of detail and lack of support in the legacy models. This service cannot be implemented at the current time. It is identified here as an indicator of direction for future development.

2.3 Reverberation

Reverberation refers to the acoustic signals produced in response to an active acoustic transmission, also known as a ping.

Reverberation consists of two parts: discrete returns and continuous background reverberation signals. These two are separated in the API because the update rates for the two kinds of information are typically different. Continuous background reverberation is more time consuming to compute but is relatively insensitive to small displacements of the transmitter and receiver. This implies the continuous background reverberation can be updated infrequently if at all.

Discrete returns are more sensitive to the location of source and receiver, and hence must usually be updated once per ping.

2.3.1 Discrete_Reverberation

Discrete returns represent both direct reception of the original signal via various paths and identifiable discrete echoes of that signal from various localized reverberation.

Service: Discrete_ Reverberation

EVA:	Get Discrete Reverb
Fey Ray:	not provided

CASS/GRAB: Not provided by GRAB. CASS has multiple reverberation models. Details could not be determined for these.

Input: source: sensor,

```
        source_location: location,
        source_orientation: orientation,
        receiver: sensor,
        receiver_location,
        receiver_orientation: orientation
Outputs: discrete_returns: sequence[discrete_return]
            // sorted in increasing order of arrival time
Exceptions: source_cannot_transmit,
              Receiver_cannot_receive
```

2.3.1.1 Type discrete_return

```
    Type discrete_return
        Create_discrete_return (time_since_ping: float, //seconds
                                departure_angle: orientation,
                                arrival_angle: orientation,
                                category: return_category,
                                amplitude: frequency_distribution);
        Get_time_since_last_ping(discrete_return): float //seconds
        Get_departure_angle (discrete_return): orientation
        Get_arrival_angle(discrete_return): orientation
        Get_category(discrete_return): return_category
        Get_amplitude(discrete_return): frequency_ distribution
    END discrete_return
```

2.3.1.2 Type frequency_distribution

```
    Type frequency_distribution IS
        Create_frequency_distribution(frequency: float            //in Hertz
                                      amplitude: float )          //-1.0 .. +1.0
                              :frequency_distribution
             //Amplitude is ratio to the signal at 1 meter from the source
            // Creates a distribution containing an amplitude at a single frequency
         Add_signal(frequency_distribution,
                    frequency: float,                             // Hertz
                     amplitude: float )                          //-1.0 .. +1.0
         Get_amplitude(frequency_distribution, frequency): float  //-1.0 .. +1.0
        // interpolates between sample frequencies throws exception
        // frequency_out_of_range  if frequency is below minimum sampled
        // frequency  or above maximum  sampled frequency
    Get_min_frequency(frequency_distribution): float             // in Hertz
    Get_max_frequency(frequency_distribution): float             // in Hertz
    Get_frequencies(frequency_distribution): sequence[float]   //in Hertz
        // the set of frequencies at which discrete amplitude samples are available

END frequency_distribution
```

2.3.2 Continuous Reverberation

Continuous background reverberation represents returns from distributed scatterers that blend together into a continuous background return pattern.

Service: Continuous_Reverberation

EVA: Get Reverb Envelopes

Fey Ray: not provided

CASS/GRAB: not in GRAB

CASS provides multiple reverberation models. Details could not be determined.

```
Inputs:  source: sensor,
         source_location: location,
         source_orientation: orientation,
         receiver: sensor,
         receiver_location,
         receiver_orientation: orientation
Outputs: reverberation: reverberation_envelope
Exception:  source_cannot_transmit,
            Receiver_cannot_receive
```

2.3.2.1 Type Reverberation_envelope

```
Type reverberation_envelope IS
    Create_reverberation_envelope
            (initial_time: float,                    //seconds
                   // delay from transmission to first received signal
             time_offset: float,                     // seconds
                   // sampling time relative to the initial time
             sample:  reverberation_envelope_sample)
```

```
        Add_sample(reverberation_envelope,
                        time_offset: float,      //seconds
                        sample: reverberation_envelope_sample)
                      // add another data sample with the specified time delay relative to
                      // initial time
         Get_initial_time(reverberation_envelope): float     // seconds
         Get_number_of_sample(reverberation_envelope): integer
                      // returns the number of sample points
       Get_sample_time(reverberation_envelope, position: integer)
                         :  reverberation_envelope_sample
                      //position in 1.. number of sample

   END reverberation_envelope
```

2.3.2.2 Type Reverberation_envelope_sample

```
   Type Reverberation_envelope_sample   IS
     Create_ reverberation_envelope_sample
          (source_beam, receiver_beam: EVA_beam_pattern,
           arrival_direction, departure_direction: float             //in degrees
           simple: frequency_distribution): reverberation_envelope_sample
            //creates a reverberation  envelope sample containing a single frequency
            // distribution, corresponding to a single source beam, receiver beam, and
            // arrival  direction
     Add_distribution(reverberation_envelope_sample,
                         source beam, receiver beam: EVA_beam_pattern,
                          arrival_direction, departure_direction: float,
                          signal: frequency_distribution)
           // add another frequency distribution corresponding to the given beam
           // pattern and angles
    Get_number_of_frequency_distributions(reverberation_envelope_sample): integer
          // returns the number of frequency distribution
         // in the reverberation envelope sample
   Get_source_beam(reverberation_envelope_sample, position: integer)
                       : EVA_beam_pattern
         // throws exception position_out_of_bounds
         // if the position is not in 1.. number_of_frequency _distributions
    Get_receiver_beam(reverberation_envelope_sample, position: integer)
                       : EVA_beam_pattern
         // throws exception position_out_of_bounds
         // if the position is not in 1.. number_of_frequency _distributions
     Get_arrival_direction(reverberation_envelope_sample, position: integer): float
                                                                              //degree
        // throws exception position_out_of_bounds
        // if the position is not in 1.. number_of_frequency _distribution
```

```
      Get_departure_direction(reverberation_envelope_sample, position: integer) : float
                                                                        //degree
         // The direction in which the signal must have departed from the transmitter to
         // arrive at the receiver with the given arrival angle
         // throws exception position_out_of_bounds
         // if the position is not in 1.. number_of_frequency _distributions
      Get_signal(reverberation_envelope_sample, position: integer)
                   : frequency_distribution
         // throws exception position_out_of_bounds
         // if the position is not in 1.. number_of_ frequency _distributions
   END reverberation_envelope_sample
```

2.4 Propagation_Loss_VS_Range

```
Service: propagation_loss_VS_Range
      // The propagation loss as a function of the range from the source, for a given
      // direction  from the source and given source and receiver beam pattern
EVA – PropagationLossVsRange
FeyRay – not provided
CASS/GRAB – details not determined

Inputs:       Source: location,
              Frequencies: band,
              receiverBearing: angle,     // in degrees
              receiverDepth: float,       // in meters
              MaxRange: float             // in meters
              Steps: integer              // determines resolution of the output
                                          // delta range = MaxRange / (Steps - 1)
              Source_beam, receiver_beam: EVA_beam_pattern

Depends on the following input attributes:
              Get_Game_Clock: time,           // represents time and date
              Get_gaming_area                 // define the part of the ocean to be simulated

Output:       PropagationLossField

Exceptions:   endpoint_outside_gaming_area(location),
              endpoint_on_land(location),
              endpoint_below_bottom(location),
              path_crosses_land,
              input_data_out_of_range,
              insufficient_memory,
                   // The API should handle buffer overflows if memory is available
              computation_failure(string)   // message states reason for failure
                   // The only generic failure exception supported by current FeyRay
```

2.4.1 Type Propagation_Loss_Field

```
TYPE PropagationLossField IS

         Create_propagation_loss_field(
                  source: location,
                  frequencies: band,
                  receiver_bearing: float,        // degrees
                  receiver_depth: float,          // meters
                  max_range: float,               // meters
                  range: float,                   // meters
                  amplitude: requency_distribution): propagation_loss_field
              // creates a propagation loss field with a single range sample
        Add_range_step(propagation_loss_field,
                           range: float,          //meters
                           amplitude: frequency_distribution)
              // add a new range step and the corresponding frequency distribution of
              // amplitude to a given propagation loss field
        Get_source_location(propagation_loss_field,
                            source_beam, receiver_beam: EVA_beam_pattern): location,
         Get_frequencies(propagation_loss_field,
                             source_beam, receiver_beam: EVA_beam_pattern): band,
        GetReceiverBearing: float,               // in degrees
        GetReceiverDepth: float,                 // in meters
        GetMaxRange, source_beam, receiver_beam: EVA_beam_pattern: float
                                                 // in meters
        GetSteps, source_beam, receiver_beam: EVA_beam_pattern: integer
                                                 // determines resolution of the output
                                                 // delta range = MaxRange / (Steps - 1)
        Get_proploss(PropagationLossField, frequency, range): float
                // Magnitude of net propagation loss
                // interpolated between sample points for frequency and range
          Exception past_maximum_range, frequency_outside_band
                // This service does interpolation only, not extrapolation.
                //  Exception frequency—outside_band is thrown if  the proploss is
                //  requested at a frequency that is not strictly  within the range of
                //  frequencies at which proploss sample points have been calculated.
        Get_proploss_samples(PropagationLossField, frequency): sequence[float]
                // The set of calculated data points, without any interpolation
                // length of sequence = Steps
                // Magnitude of net propagation loss,
                // sampled at equally spaced ranges in [0 .. MaxRange]
                // The sampled frequencies for the frequency input parameters can be
                // obtained via get_frequencies.

END PropagationLossField
```

2.5 Point_to_Point_Propagation_Loss

```
Service: point_to_point_propagation_loss
// Propagation loss from a single source to a single receiver. This service calculates a set
// of eigen paths connecting the source and receiver and an aggregate propagation loss
// value that combines the contributions from all of the paths. The attributes calculated
// for each path, include propagation loss values, phases, travel times. Etc. See the
// description of TYPE path below for details.

EVA – PropagationChannelData
FeyRay – output in KINGKONG data structure, service name unknown, inputs unknown
CASS/GRAB – COMPUTE GRAB EIGENRAYS

Inputs:     Source: location,
            Receiver: location,
            Frequencies: band

Depends on the following input attributes:
            Get_Game_Clock          // represents time and date
            Get_Gaming_Area         // defines the part of the ocean to be simulated
 Affected by the following EVA control attributes:
            Set_MaxPaths: integer,
                // EVA-specific optional upper bound on output size
                // Default should be larger than expected worst case
                // approximately 250 ?
            Set_MaxFinePaths: integer
                // EVA-specific optional upper bound on fine structure size
                // Default zero if not specified, means do not compute fine paths

Outputs:    ps: set[path]
            Combined_proploss: float    // in decibels
                        //  The noncoherent sum of the  propagation losses over
                        //  all the paths, at the frequency with  the smallest loss.
Exceptions: endpoint_outside_gaming_area(location),
            endpoint_on_land(location),
            endpoint_below_bottom(location),
            path_crosses_land,
            input_data_out_of_range,
            insufficient_memory,
                // API should handle buffer overflows if memory is available
            computation_failure(string)   // message states reason for failure
            // The only generic failure exception supported by current FeyRay
```

2.5.1 Type path

```
TYPE path IS
    Get_id(path): integer,                      // unique id for each path
```

```
 Get_source_location(path): location,
 Get_receiver_location(path): location,
 Get_frequencies(path): band,
 Get_departure_direction(path): vector 3;            // unit vector: length=1.0
          // vertical angle at the source is most significant
 Get_arrival_direction(path): vector 3;              // unit vector: length=1.0
          // vertical angle at the receiver is most significant
 Get_time_delay(path): time                          // in seconds
          // The propagation time from the source to the receiver along the current
          // path. Can be used for calculating phases of signals for coherent
          // combinations (Specifically needed to get accurate phase information in
          // EVA, see comment on Get_proploss ).
 Get_delay_gradient_at_source(path): vector 3        // in seconds per meter
          // The gradient of the time delay with respect to the position of the
          // source—a three dimensional vector 3 of derivatives.  Currently EVA
          // calculates only derivatives in the radial and vertical directions.
          // For transverse displacements much smaller than the range, the derivative
          // in  the transverse direction is approximately  equal to the derivative in
          // the radial direction, times x/r, where x is the transverse
          // displacement and r is the distance from source to receiver. See Fig.5
          // This approximation can be used to compute the third component of the
          //  derivative that EVA does not supply.
          // Gradients are not supported by current version FeyRay, but comments
          // identify them as a possible future FeyRay capability
 Get_delay_gradient_at_receiver(path): vector 3      // in seconds per meter
          // The gradient of the time delay with respect to the position of the
          // receiver—a three dimensional vector 3 of derivatives. Currently EVA
          // calculates only derivatives in the radial and vertical directions. For
          // transverse displacements much smaller than the range, the derivative in
          //  the transverse direction is approximately  equal to the derivative in the
          // radial direction, times x/r, where x is the transverse
          // displacement and r is the distance from source to receiver. See Fig. 5
          // This approximation can be used to compute the third component of the
          //  derivative that EVA does not supply.
          // Gradients are not supported by current version FeyRay, but comments
          // identify them as a possible future FeyRay capability
 Get_proploss(path, frequency): complex
Exception frequency_outside_band
          // interpolated between sample points for frequency
          // phase depends on path length and number of reflections and has to be
          // calculated from the time delay in EVA to give accurate results
 Get_frequency_spreading(path, sampling_rate: float): random_delay
          // Sampling rate is in Hertz, see random_delay.Get_delay_sample .
          // Models path length variation due to waves, see EVA user's guide
          // appendix A and appendix 2 of this report
          // Smith: this is a rough approximation, should depend on frequency
```

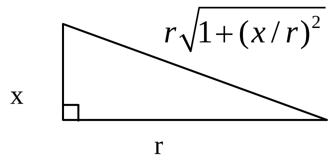
```
        Get_fine_paths(path, sampling_rate: float): set[random_delay]
                // Sampling rate is in Hertz, see random_delay.Get_delay_sample .
                // Fine paths represent variations in the propagation delay due to boundary
                // interactions resulting in multiple paths, with slight different travel times.
                // Examples of such interactions include reflections from different layers
                // of sediment, non-specular reflections from a rough bottom or a rough
                // surface, and multiple paths in a surface duct. The fine path
                // approximation appears to be specific to EVA, see user's guide EVA
                // appendix B and appendix 2 of this report.

END path
```

If the speed of sound is constant:

(A) in the radial direction

r=ct, t=r/c, dt /dr= 1/C

(B) in the transverse direction



$t = (r/c)\ \mathrm{SQRT}\ (1+(x/r)^2)$

$dt / dx = (r/c)(2x/ r^2) / 2\mathrm{SQRT}\ (1+(x/r)^2)$

$= (x/rc) / \mathrm{SQRT}\ (1+(x/r)^2)$

$\approx (x/rc) = (x/r)(dt/dr)$ if $x<<r$

This approximation should hold for small transverse displacements even if speed of sound varies and paths are slightly curved.

Fig. 5 Approximate Transverse Derivative of Travel Time

2.5.2 Type Location

```
TYPE location IS
        Create_location(lat, long, depth: float): location  // degrees, degrees, meters
        Get_ longitude(location): float,                     // degrees
        Get_ latitude(location): float,                      // degrees
        Get_depth(location): float,                          // meters
        Distance(l1, l2: location): float                    // meters
                                                             // straight line distance between the
                                                             // two locations
END location
```

2.5.3 Type vector 3

```
TYPE vector 3 IS        // 3-D vectors
        Cartesian_vector 3(x, y, z: float): vector 3,
                // Generic constructor using Cartesian coordinates
```

```
            // can be used with any kind of units
            // (e.g. see Get_delay_gradient_at_source)
    Displacement_vector 3(start, end: location): vector 3
            // The vector 3 from start to end, in units of meters
    Length(vector 3): float                  // The magnitude of the vector
    Normalize(vector 3): vector 3            // unit vector parallel to the original
    Plus(vector 3, vector 3): vector 3       // vector  sum
    Minus(vector 3, vector 3): vector 3      // vector  difference
    Times(float, vector 3): vector 3         // scalar multiplication
    Dot(vector 3, vector 3): vector 3        // dot product
    Cross(vector 3, vector 3): vector 3      // cross product
          // also methods for getting components of a vector  with respect to
          // various coordinate systems, details TBD

END vector 3
```

2.5.4 Type complex

```
TYPE complex IS       // complex numbers
    // Introduced to represent magnitude and phase of propagation loss
    // constructors:
    Cartesian_complex(r, i: float): complex,
            // Generic constructor using the real and imaginary part
            // can be used with any kind of units
            // matches FeyRay
    Polar_complex(magnitude, phase: float): complex
            // The magnitude is the absolute value of a dimensionless ratio,
            // The angle is in degrees
            // matches EVA
    // Attributes and operations:
    Real_part(complex): float
            // The real component of a complex number
    Imaginary_part(complex): float
            // The imaginary component of a complex number
    Magnitude(complex): float                // The radius, wrt polar coordinates
            // if c = a + b*i then Magnitude(c) = sqrt(a*a + b*b) , a and b are real
    Phase(complex): float                    // The angle, wrt polar coordinates
            // if c = a + b*i then tan(phase(c)) = b/a, a and b are real
    Plus(complex, complex): complex          // complex sum
            // Complex sums are used for coherent combinations of signals from
            // different paths that account for phase differences
            // if Re(c)= Real_part(c) and Im(c) = Imaginary_part(c)
            // then Re(x + y) = Re(x) + Re(y), Im(x + y) = Im(x) + Im(y)

    // The following are standard operations on complex numbers –
    // not clear if they are all needed for the acoustic API,
    // they are included to support future applications
    Minus(complex, complex): complex            // complex difference
            // if Re(c)= Real_part(c) and Im(c) = Imaginary_part(c)
            // then Re(x - y) = Re(x) - Re(y), Im(x - y) = Im(x) - Im(y)
    Times(float, complex): complex              // complex multiplication
            // if Re(c)= Real_part(c) and Im(c) = Imaginary_part(c)
            // then Re(x * y) = Re(x) * Re(y) – Im(x) * Im(y)
            // Im(x * y) = Re(x) * Im(y) + Im(x) * Re(y)
    Divide(complex, complex): complex           // complex division
            Exception division_by_zero
            // if Re(c)= Real_part(c) and Im(c) = Imaginary_part(c)
            // then Re(x / y) = (Re(x)*Re(y) + Im(x)*Im(y))/(Re(y)^2 + Im(y)^2)
            // Im(x / y) = (Im(x)*Re(y) - Re(x)*Im(y))/(Re(y)^2 + Im(y)^2)
    Conjugate(complex): complex                 // same magnitude, negative phase
            // if Re(c)= Real_part(c) and Im(c) = Imaginary_part(c)
            // then Re(Conjugate (c)) = Re(c), Im(Conjugate (c)) = - Im(Conjugate (c))
END complex
```

2.5.5 Type band

```
TYPE band IS
    Create_EVA_band(min_ frequency, max_frequency: float,
            ratio: enumeration[octaves, thirds]): band
        // model specific constructor
        // EVA allows only geometrically spaced samples
        // with ratio = 2.0 (ratio = octaves) or 2.0^(1/3) (ratio = thirds)
    Create_FeyRay_band(min_ frequency: float): band
        // FeyRay allows only linearly spaced samples that are multiples
        // of the minimum frequency.
        // max_frequency = number_of_samples * min_frequency
        // The number of samples in FeyRay is a constant fixed at compile time,
        // currently 512, may have to be a power of 2 to support FFT.
    Get_min_frequency(band): float      // Hertz
    Get_max_frequency(band): float      // Hertz
    Get_sample_frequencies(band): sequence[float]
        // The sample frequencies in the band
END band
```

2.5.6 Type random_delay

```
TYPE random_delay IS
    EVA_frequency_spreading_delay(RMSDelayVariation: float,
        DelayVariationFrequency: float,
        DelayVariationBandwidth: float,
        ZeroToPeakRatio: float,
        Sampling_rate: float) : random_delay         // constructor matching EVA
        // stochastic model of Bretschneider spectrum for random path length
```

```
                    // fluctuations from surface reflections due to large scale waves.
                    // Sampling_rate is the frequency in Hertz at which Get_delay_sample
                    // will be invoked.  EVA supports rates in the range [5.0 .. 12.0].
          EVA_fine_path_delay(Amplitude: float;
                    FineDelay: float;
                    Absorption: float;
                    PatchWidth: float;
                    PatchLength: float;
                    PatchOffset: float;
                    Sampling_rate: float): random_delay // constructor
                    // matching EVA fine paths, see EVA appendix B.
          Get_delay_sample(random_delay): float               // in seconds
                    // The random delay samples are added to a constant proploss delay
                    // to get a realistic proploss time series at the specified sampling rate.
                    // Samples from random delays constructed using
                    // EVA_frequency_spreading_delay or EVA_fine_path_delay
                    // must be called periodically at the specified sampling rate,
                    // where the rates are relative to the game clock.

END random_delay
```

3 INTITIALIZATION / INPUT SERVICES AND ATTRIBUTES

3.1 Implicitly defined attribute Access Functions

Every settable attribute has an implicitly defined readout operation — see section 2.1 for detail.

3.2 EVA Initialization

```
TYPE EVA_acoustic _API  IS
         Create_EVA_acoustic_API(EVA_initialization_parameters,
                                                EVA_gaming_parameters,
                                                EVA_environment_parameters)
                    // Initialize the server using a randomly generated password that is stored
                    // internally in the API object, and initialize the models, gaming area,
                    // and simulated environment
        Set_game_clock(EVA_acoustic_API, integer) //in seconds from the starting date,
        default 0
        Set Clutter Density(EVA_acoustic_API, float)
                   // average number of returns per square meter of ocean area
                   // represents density of random clutter returns in addition to seamounts
         Set Clutter Persistence ( EVA_acoustic_API, float) //in seconds
                   // average persistence time of each random clutter return
         Set Global Wind Speed (EVA_acoustic_API, float, integer)
                  // speed in meters per second, transition time in seconds
          Set Global Rain Rate (EVA_acoustic_API, float, integer)
                  // Rain rate in mm per hour, transition time in seconds
           Set_local_rain_shower (EVA_acoustic_API,set[EVA_rain_shower], integer)
                  //set of storm descriptions, transition time in seconds
                  // period over which previous storms will fade out and
                 // the new ones will fade in.
           Set_local_biological_reverberation(EVA_acoustic_API,
                                                              set[EVA_biological_reverberator],integer)
              // set of biological reverberation areas, transition time in seconds
           Set_sensor(EVA_acoustic_API, EVA_sensor, orientation, location, game_clock:
                          integer )
              // corresponds to both deployActiveAssets and updateActiveAssets in
              //EVA. The is-deployed attribute of a sensor keeps track of which EVA
              // operation is needed.
END  EVA_acoustic_API
```

3.2.1 Type EVA_initialization_parameters

```
TYPE EVA_initialization_parameters

         Create_EVA_initialization_paramters()

        // optional performance tuning parameter follow
```

```
    Set_PL_processor_share (EVA_initialization_parameter, float) //0,1,0.33
    Set_Noise_processor_share(EVA_initialization_parameter, float) //0,1,0.33
    Set_Reverb_processor_share (EVA_initialization_parameter, float) //0,1,0.33
    Set_min_frequency(EVA_initialization_parameter, integer) // in Hertz
    Set_max_ frequency(EVA_initialization_parameter, integer) // in Hertz
    Set_min_active_frequency(EVA_initialization_parameter, integer) // in Hertz
    Set_max_active_frequency(EVA_initialization_parameter, integer) // in Hertz
    Set_min_sensor_depth(EVA_initialization_parameter, integer) //in meters
    Set_max_sensor_depth(EVA_initialization_parameter, integer) //in meters
    Set_sensor_types(EVA_initialization_parameter, set[sensor_type])
    // supply reasonable default values for all optional attribute.

END EVA_initialization_parameters
```

3.2.2 Type EVA_gaming_parameters

```
TYPE  EVA_gaming_area IS
     Create_EVA_gaming_area(
             lat,long: float,          //center point of area, in degrees
             Height, width: float,   //NS and EW size of gaming area, in meters
             Start: date)
      Set_max_ships (user's guide, integer)
      Set_max_ seamounts (EVA_gaming_area, integer)
      Set_game_length(EVA_gaming_area, integer)    //in seconds
      Enable_EVA_position_TL(EVA_gaming_area ,Boolean)
      // enables and disables approximation that speeds up reverberation at
      // the expense of accuracy, default is disabled.
      // The implementation must set safe and reasonable default values for all
      // optional parameters.
 END EVA_gaming_area
```

3.2.2.1 Type EVA_date

```
TYPE EVA_date IS
      Create_date(month, day, hour, min, sec: integer): date
            //use GMT for time of day
       Get_month(EVA_date): integer  //1..12
       Get_day (EVA_date): integer  //1..31
       Get_time_of_day(EVA_date): integer  //seconds since midnight
            // This operation supports the connection to EVA.
 END EVA_date
```

3.2.3 Type EVA_rainshower

```
TYPE EVA_rain_shower IS
     Create_EVA_rainshower(offset: vector2, size: float, windspeed: float,
                                     rain_rate: float)
       //offset is the displacement from the center of the gaming area
     Get_offset (EVA_rainshower): vector 2
```

```
        Get_size(EVA_ rainshower): float  //in meters
         Get_wind_speed(EVA_rainshower): float
         Get_rain_rate(EVA_rain_shower): float  // m m per hour
   END EVA_rain_shower
```

3.2.4 Type EVA_biological_reverberator

```
    TYPE EVA_biological_reverberator IS
        Create _biological_reverberator(offset:vector 2,size: float,
                                              Scatteringstrength: float);
          // size in meters, scattering in dB per square meter
        Get_offset(EVA_biological_reverberator): vector 2
        Get_size( EVA_biological_reverberator): float //meters
        Get_scattering_strength(EVA_biological_reverberator): float //mm per hour
   END EVA_biological_reverberator
```

3.2.5 Type Vector2

```
         // Two dimensional vectors used to describe positions in the gaming area

   TYPE vector 2 IS
    Create_displacement_vector2(x,y: float) :vector
         //x is month offset in meters; y is east offset in meters
   Get_x(vector 2): float  //meters
   Get_y (vector 2): float  //meters
  ENS vector 2
```

3.2.6 Type EVA_sensor

```
  TYPE EVA_sensor  IS                              //represents a receiver or transmitter
   Create_EVA_sensor(EVA_sensor_type): sensor
   Get_sensor_id(EVA_sensor): integer              // a unique id
   Get_sensor_type(EVA_sensor): sensor-type
   Set _is_deployed (sensor, boolean)

        //True if the sensor has already been deployed in the simulation.
       //Supports EVA  bookkeeping
  END EVA_sensor
```

3.2.7 Type EVA_sensor_type

```
     TYPE sensor_type IS
        Create_sensor_type(set(frequency), set(beam)): sensor_type
        Get_sensor_type_id(sensor_type): integer // unique id
        Get_frequencies(sensor_type): set(frequency)
        Get_beams(sensor_type): set(beam)
        Get_min_frequency(sensor_type): float
        Get_max_frequency(sensor_type): float
    END sensor_type
```

3.2.8 Type Orientation

```
TYPE Orientation IS
  Create_orientation (tilt_angle, tilt_direction:float) : orientation
   Get_tile_angle (orientation): float
   Get_tile_direction (orientation): float
END  Orientation
```

3.2.9 Type EVA_Beam

```
TYPE EVA_Beam IS
    Create_Beam(mode: xmit_receive_mode, frequency: float, beam_pattern): beam
      // create a beam with a single frequency
      //throws exception useless_beam if xmit and receive are both false
   Add_beam_pattern(EVA_beam, frequency, beam_pattern)
     // add the specified beam pattern and frequency to the beam, frequency in Hertz.
    Get _mode(EVA_beam): xmit_receive_mode
    Get_frequency(EVA_beam): frequency[float]
    Get_beam_pattern(EVA_beam, frequency): beam_pattern
      //throws exception ,no such frequency is the frequency isnot a member of
      //get_frequencies
END EVA_Beam
```

3.2.9.1 Type EVA_Beam_Pattern

```
TYPE EVA_ Beam_Pattern
    Create_Beam_pattern(able: float, level: flaot)
          //angle in degrees, level in dB
         //create a pattern with only one direction
   Add_direction(beam_pattern, angle: float, level: float)
         //add the specified direction and level to the beam pattern
   Get_beam_pattern_id(beam_pattern): integer   // a unique id
   Get_angles(beam_pattern): sequence(float)
         //returns the sequence of angles with explicitly stored level
    Get_level(beam_pattern, angle: float): float
          // returns the level at the given angle, in dB, interpolating if necessary
          //throws exception angle_out_of_range if the angle is outside the range of
          // explicitly stored angles
END Beam_Pattern
```

3.2.10 Type Xmit_receive_mode

```
TYPE Xmit_receive_mode  IS
    Enumeration [receive, transmit, both]
 END  Xmit_receive_mode
```

3.3 Fey Ray Initialization

```
TYPE  Fey_Ray_Acoustic_API  IS
    Create_Fey_Ray_Acoustic_API(…)
     //details of possible parameters unknown
END Fey_Ray_Acoustic_API  IS
```

3.4 CASS /GRAB Initialization

```
  TYPE  CASS_GRAB_Acoustic_API  IS
      Create _CASS_GRAB_Acoustic_API(…)
           // details of possible parameters unknown
END CASS_GRAB_Acoustic_API
```

4 PERFORMANCE TUNING ATTRIBUTES

4.1 EVA performance Tuning

The performance tuning parameters for EVA are the following:

Set_EVAPModels
Set_EVANModels
Set_EVARModels
Set_MinFreq
Set_MaxFreq
Set_MinActiveFreq
Set_MaxActiveFreq
Set_MinSensorDepth
Set_MaxSensorDepth
Set_MaxRam
Set_MaxDisk
Set_MaxPaths
Set_MaxFinePaths

These correspond directly to parameters in the underlying model.

4.2 Fey Ray Performance Tuning

The performance tuning attributes for Fey Ray are the following:

Set_Ray_Trace_Bounds
Set_allow_Retrograde_Ray_Trace
Set_TL_Max
Set_Max_Sur_Refs
Set_Max_Bot_Refs
Set_Step_Size_Lims
Set_Ray_Trace_Tols

These correspond directly to parameters in the underlying model.

4.3 CASS /GRAB Performance Tuning

Details unknown

5 OPEN ISSUES REGARDING THE API

At the current time there are no accepted standards for modeling active sonar.
As best we could determine, Fey Ray does not provide any services, but there does not appear to be any way to simulate an echo return from a platform that is not transmitting. The only, kinds of scatters that can be modeled via the EVA interface are seamounts, random surface clutter, and random biological reverberation. CASS/GRAB has a variety of reverberation models, which the current study did not evaluate in detail due to lack of time.

Since the purpose of the API is to simulate sensor outputs to train sonar operations, and real sensors provide a stream of acoustic pressure measurements, we would expect the most natural service to be calculation of a pressure time series for a given sensor. Neither Fey Ray nor EVA directly support such a service, although one could be synthesized from the reverberation services provided by EVA.

CASS/GRAB have facilities for computing acoustic pressure—see COMPUTE PRESSURE (CMP_PRS). However, we were unable to extract details of the interface to this services from available documents.

Appendix 1 - Summary of known information about the CASS/GRAB interface

1. Introduction

CASS is a very complicated system that provides access to a large number of software models related to ocean acoustics. Only one of those, GRAB, is considered to be a Navy standard. This appendix therefore focuses on information relevant to defining an API for GRAB. Precise information about the interfaces has been impossible to obtain. The relevant information we did obtain is summarized below.

2. Quotes describing the GRAB interface

The following is extracted from [1]:

The nominal inputs to the GRAB model include a description of the environment, source and receiver geometry, acoustic frequencies and a user-specified fan of test rays. GRAB is dimensioned to efficiently accommodate multiple frequency calculations. The GRAB environment consists of independent, two-dimensional range-dependent surface, bottom and ocean grids. The bottom grid accommodates the bathymetry and the surface grid allows for deterministic surface topology. Range-dependent sound speed, temperature, salinity, surface conditions and bottom composition populate the ocean environment grid.

Primary outputs of the GRAB model include three binary files and an ASCII file that lists all the user inputs and prints the pressure and eigenrays. One of the binary files is an eigenray file containing the range, source angle, target angle, travel time, phase, amplitude and path history for the eigenrays for the specified source and target depth. The other two files are pressure files. The first is the pressure file of coherent or random propagation loss for the specified source and receiver geometry. The other is the contour file of the pressure field as a function of depth and range.

The following is extracted from [2]:

The general program flow of the OAML CASSv3 model is illustrated in the following figure.

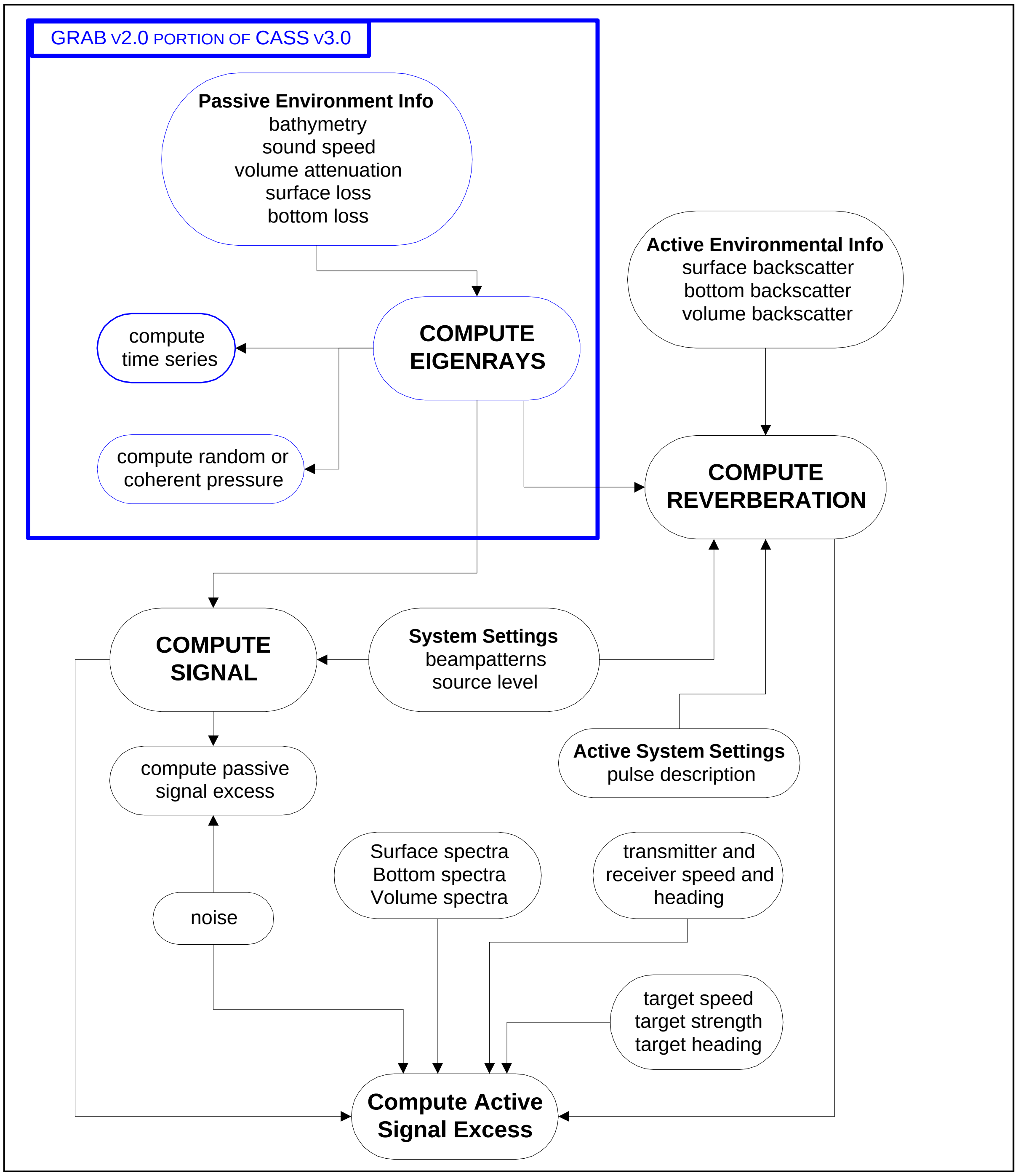


Figure 4.2-1. The General Program Flow of the OAML CASSv3 Model

The portion of CASS that is highlighted in the box (passive environment information, COMPUTE EIGENRAYS, compute time series, compute random or coherent pressure) represents that portion of CASS that comprises the GRAB eigenray model. CASS includes the system source level and beam pattern information to generate the signal information from the GRAB eigenrays. CASS incorporates the system beampattern, source level, pulse length information and environmental backscatter, and spectral models with the eigenrays to compute reverberation. The portion of CASS that combines the signal, reverberation, target, and noise to compute active signal excess is not configuration managed.

The following is extracted from [3]:

Output of the GRAB model can include three binary files and an ASCII file. GRAB echoes the user inputs on the screen and this information can be redirected to an output file. Two of the binary files are eigenray files; EIGENRAY.DAT and EIGENRAY.EIG. The EIGENRAY.DAT file is a header file for the EIGENRAY.EIG file that contains the actual eigenray information. The other file is the pressure file of coherent or random propagation loss for the specified source and receiver geometry. See the Software Design Description[3] for the specific file formatting.

The software design description reference in footnote 3 cited in the above quote refers to: Keenan, R., H. Weinberg and F. Aidala, Jr., "Software Design Description: Gaussian Ray Bundle Eigenray Propagation Model" OAML-SDD-74, Stennis Space Center, MS: March 1999. We did not have access to that document. However, reference [1] appears to be a later version of that document, and the excerpt quoted above is the only description of binary file contents we could find there. That information is not detailed enough to serve as the basis for a well-grounded API design.

3. Assessment of CASS system structure, documentation and feasibility of a CASS API

The CASS system is implemented mostly in FORTRAN 77 and has a corresponding old-style structure that is based on subroutines and is tightly coupled by data dependencies. The system has a command language oriented toward punch card style batch processing. There are no encapsulated subcomponents or subsystems in the modern sense, and there is no data encapsulation. Common blocks and global variables are used heavily.

CASS has voluminous documentation (many thousands of pages) that is mostly oriented towards implementers and is largely unintelligible by potential users. The main documents appear to be a reference guide, a software requirements description, a software design description, and a software test description. We were unable to find even a specific reference to a user's guide for CASS, let alone the desired document itself.

The CASS documentation contains a great deal of detail about the internal operation of the algorithms and the mathematics on which they are based. There is very little information about the meaning of the command language (format is specified in detail, however), or about the data that must be supplied by a potential user, its structure and intended meaning, what the commands do, how they relate to each other, what system concepts the users must know in order to understand how to use the commands and how to make them work together, and how to interpret the meaning of the system outputs. It is also lacking in good descriptions of the structure, intended meaning, and interrelationships of the data in the system. The missing information is vital for the design of a sound and reliable API. The existing commands are designed to be used by a person, not by another program. There is a list of commands but not a coherent explanation of what each command will do for a user.

To develop a good API for CASS, it will be necessary to first develop a document containing the missing information identified above, and then to reengineer the system to encapsulate the individual services and to make them accessible to external software applications. That would be a major undertaking. Success would require a minimum of a one person-year dedicated commitment from the developers of CASS, together with ten to thirty person-years from skillful software engineers just to obtain accurate service-level documentation, and a larger subsequent effort to do the re-engineering. The dedicated commitment and substantial participation by the developers would be vital – without that, even thirty person-years might not be enough to reconstruct and validate the missing information about the data, its intended interpretation, the implicit constraints on the data, and the relationships and constraints that relate the services, all of which are needed both for the documentation and for the reengineering. The system is quite large, so that reconstructing constraints and relationships implies a great deal of searching, cross correlating, and analysis.

The main strength of CASS, especially the GRAB part, is the fact that the accuracy of the results has been validated against real ocean measurements. However, there are currently major barriers to its widespread use to support multiple applications such as training simulations via a portable API. The main problem is the time and cost that would be required to remedy the shortfalls identified above.

4. References

[1] Ruth Keenan, Henry Weinberg, Frank Aidala, SOFTWARE REQUIREMENTS SPECIFICATION GRAB: Gaussian Ray Bundle Eigenray Propagation Model, July 30, 1997
[2] Ruth Keenan, Denise Brown, Emily McCarthy, Henry Weinberg, Laurie Gainey , Gary Brooke, Software Design Description for the Comprehensive Acoustic System Simulation (CASS Version 3.0) with the Gaussian Ray Bundle Model (GRAB Version 2.0), NUWC-NPT Technical Document 11,231, 1 June 2000
[3] Ruth Eta Keenan, Henry Weinberg, Frank E. Aidala, Jr., SOFTWARE TEST DESCRIPTION GRAB: Gaussian RAy Bundle (GRAB) Eigenray Propagation Model Version 2.0 ,June 2000

Appendix 2 - Analysis of EVA Model Signal Fluctuations due to Boundary Interactions

The purpose of this section is to describe the basic physics of the phenomena associated with the algorithms described in Appendices A & B of the EVA User's Guide. These are related to the frequency spreading and fine paths output of the point_to_point propagation loss- see the type "path" in section 2.5.1. Each deals with time variability of the signal along a single multipath. Although one uses the terminology "frequency spreading" while the other uses the term "time-delay spreading" to characterize the effects, the two phenomena are actually quite similar and have a common physical cause. Both are generated by rough interface scattering, and both involve temporal and frequency spreading.

Appendix A treats the spreading due to path interactions with a dynamically evolving rough surface. A static rough surface can create facet reflections, which can generate multiple arrival paths at nearly the same arrival angle but slightly different travel times. These arrivals follow nearly identical paths, and thus interact with a common "patch" of sea surface. Additionally, the surface is a dynamic interface, with a shape that evolves and continually changes facets. This introduces a time variability in the signal which includes Doppler shifts due to scattering from different facets moving at different speeds and different directions.

Fortunately, the dynamic nature of the sea surface has been well studied, and there exist empirical spectra that do a reasonable job of fitting measured data. Specifically, the developers chose to use the Bretschneider spectrum, which requires four parameters to characterize the mean spectrum: RMSDelayVariation, DelayVariationFrequency, DelayVariationBandwidth, and ZeroToPeakRatio. The first parameter defines the rms variation (in secs) around the mean time delay of the path. The second defines the frequency (in Hz) of the spectral peak. The third gives the 3dB down width of the spectral peak, and the fourth provides a normalizing factor that relates the power spectral level at zero frequency to that at the peak. (Formally, the power spectral level at zero frequency should be zero. But this non-zero value is used to avoid singularities.)

Although the developers assume such a spectrum will be utilized, the actual spectrum is up to the discretion of the user, as they must generate code outside of EVA to compute the actual travel time fluctuations. In that case, the user must utilize the same four output parameters to characterize the fluctuations. In Appendix A, the developers provide a description of an algorithm for approximating the Bretschneider spectrum and generating random time delays.

Figure 1 displays the data flow model, indicating where the data comes from and how it is to be utilized.

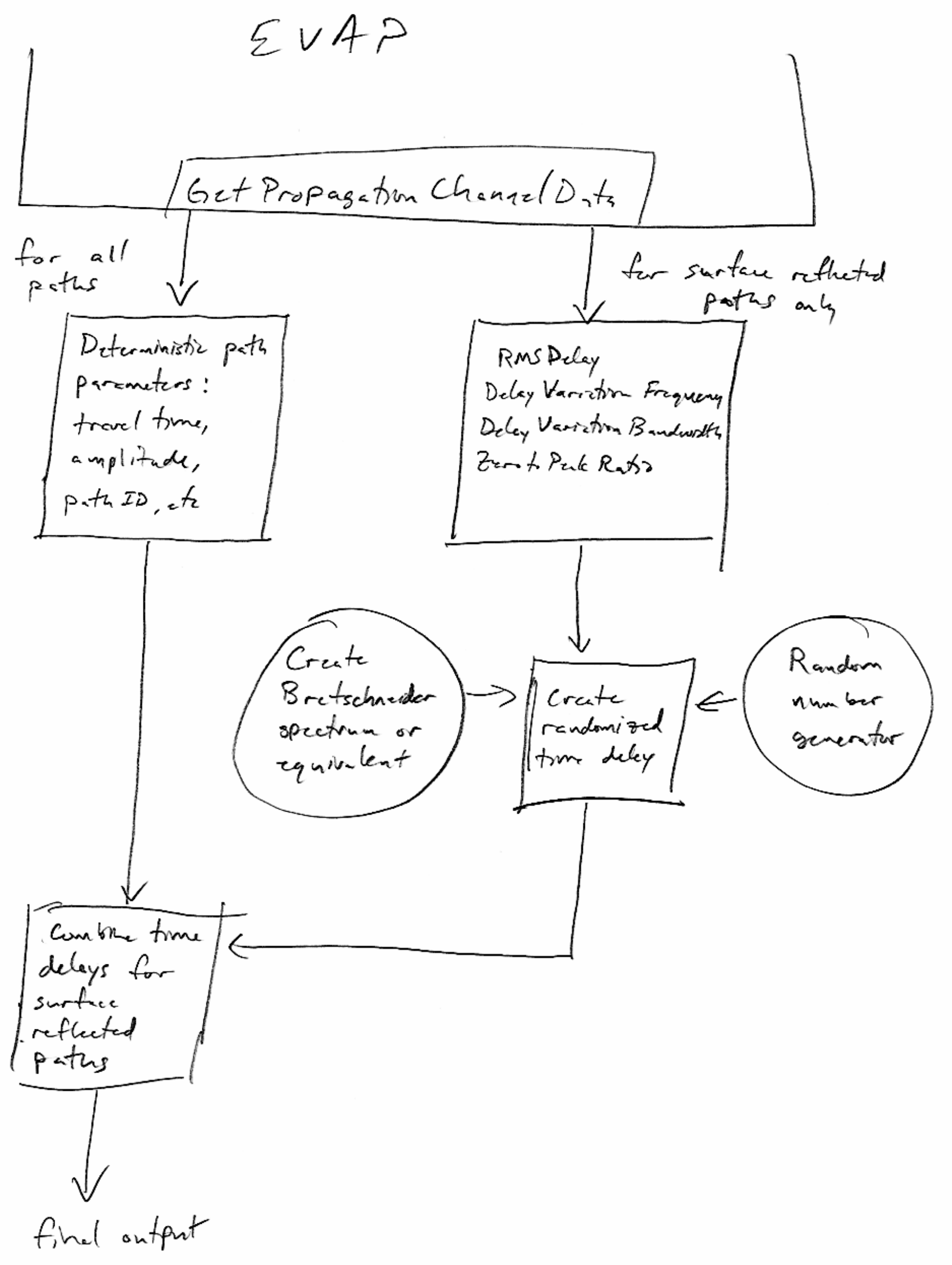

EVAP
Get Propagation Channel Data
for all paths
for surface reflected paths only
Deterministic path parameters: travel time, amplitude, path ID, etc
RMS Delay
Delay Variation Frequency
Delay Variation Bandwidth
Zero to Peak Ratio
Create Bretschneider spectrum or equivalent
Create randomized time delay
Random number generator
Combine time delays for surface reflected paths
final output

Figure 1. Data flow for random travel time fluctuations as described in EVA Manual Appendix A.

Appendix B describes the temporal spreading associated with other, more generic, types of boundary interactions. The three interactions described explicitly (which do not necessarily account for all such interactions) can most easily be understood by examining the sample paths displayed in Fig. 2.

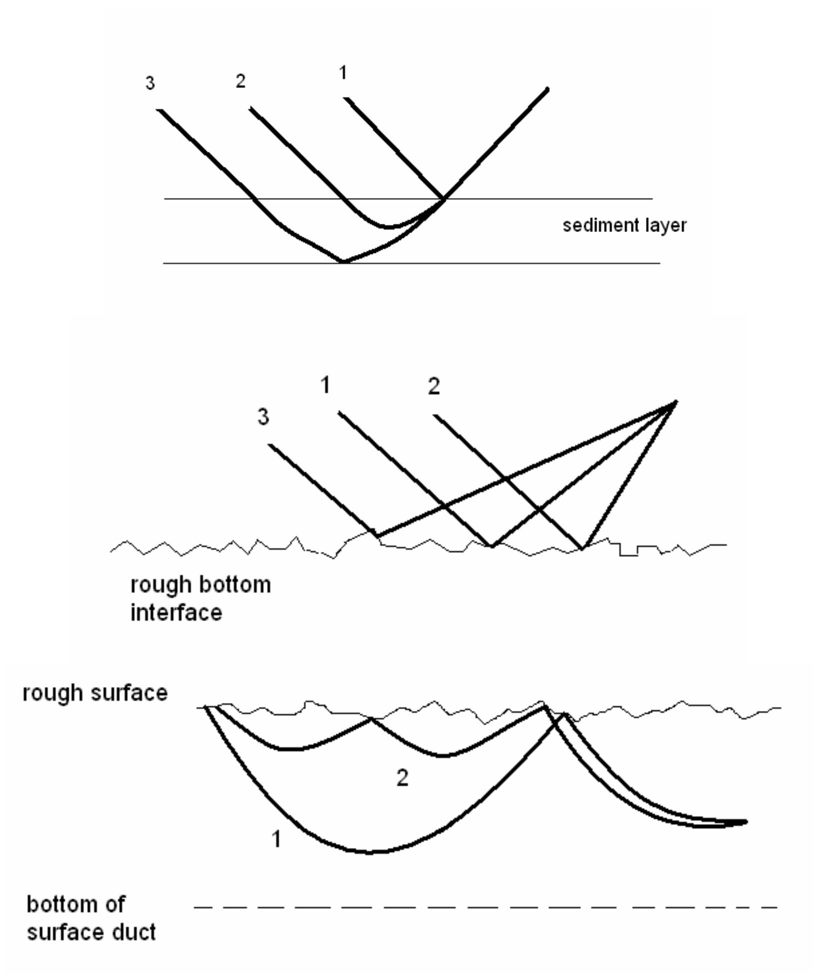


Figure 2. Physical sketches of multipath phenomena associated with time-delay spreading, as described in Appendix B of EVA manuscript.

In each of the scenarios described, an arrival along a particular direction (within the resolution of the system) can actually be comprised of several multipaths, each having a slightly different travel time. These separate arrivals are characterized as “fine paths” associated with the standard path, which is to say that the code only generates the standard path from specular reflections, and then uses a separate algorithm to compute details of the fine path structure around the mean. As in the previous calculation, the mean properties of the path are computed and output to the user, who must then separately compute the influence of any fine path time delays.

Because the phenomena described above do not fit any standard spectral shape (like the surface scatter in Appendix A), the variability is not as well defined and requires more processing by the user outside the code. For each mean path computed, EVAP also reports four parameters associated with all the fine paths: NFinePaths, SourceVirtualDepth, ReceiverVirtualDepth, and FineDelayCorrelationDistance.

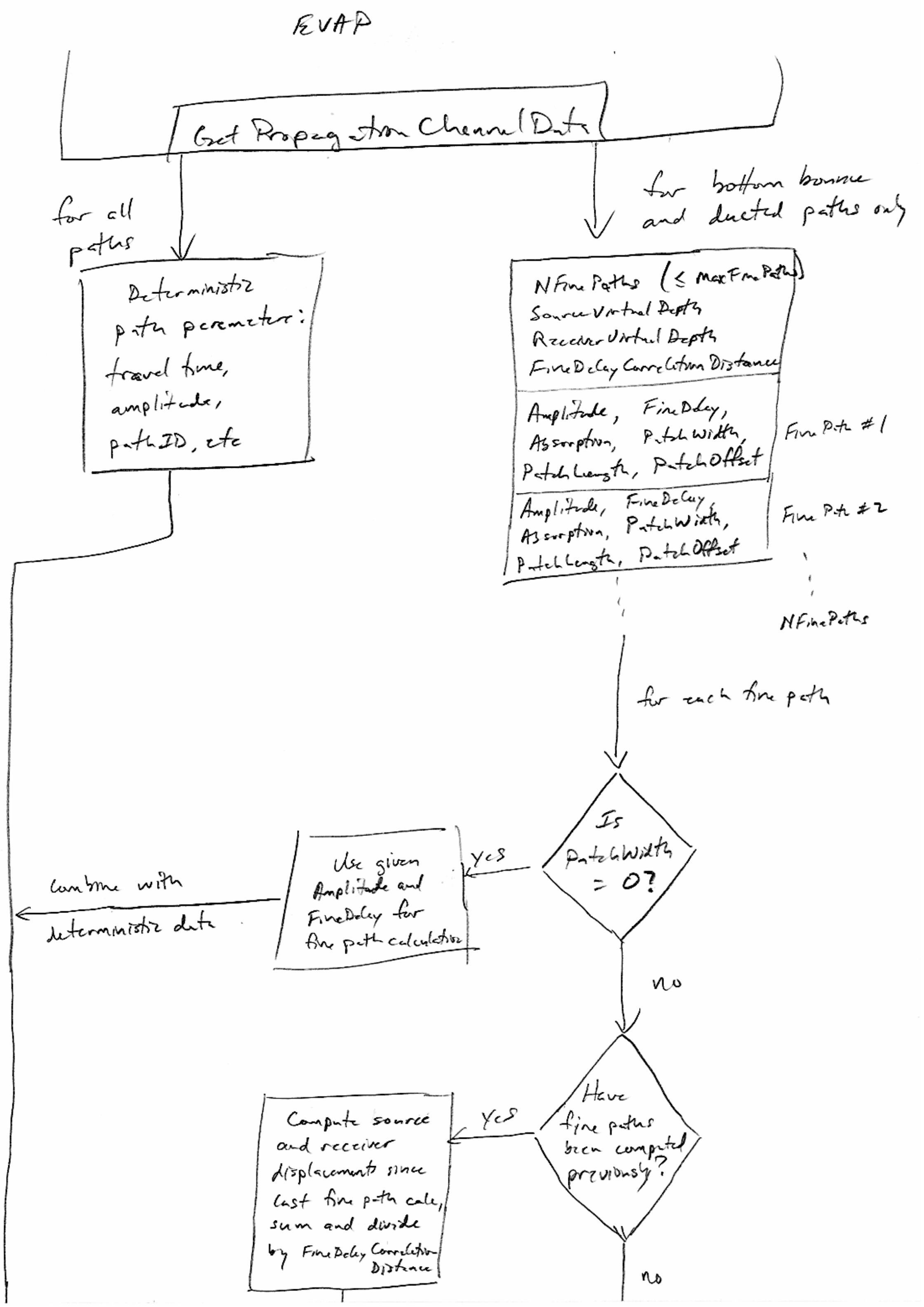
If NFinePaths is zero, no fine path data is computed, and the only path considered is the standard ray path. If NFinePaths is greater than zero (cannot exceed MaxFinePaths defined by user in call to GetPropagationChannelData), then for each fine path a six-word record is included in the model output. The parameters defined by this six-word record are: Amplitude, FineDelay, Absorption, Patchwidth, PatchLength, and PatchOffset.

The first fine path value of FineDelay is always zero, indicating the first arrival of energy on the path. Furthermore, the amplitudes are normalized such that the sum of their squares is unity. This is computed for the lowest frequency specified in the EVAP call to GetPropagationChannelData. Relative amplitudes for other frequencies can be estimated using the Absorption term, which gives the frequency dependence of the loss on the fine path (in dB/kHz).

If Patchwidth is zero, the time delay and amplitude of the fine path are taken directly from Amplitude and FineDelay. If Patchwidth is not zero, however, both the time delay and amplitude are randomized. The EVA manual describes a method for generating random values for amplitude and time delay based on a physical model of scattering from a finite size patch of interface. Specifically, the parameters PatchWidth, PatchLength, PatchOffset, SourceVirtualDepth, and ReceiverVirtualDepth are combined with a random number generator to produce variations in the time delay and amplitude of the fine path.

Finally, changes in source and/or receiver positions for each calculation are compared to the FineDelayCorrelationDistance to determine if new fine path data is needed.

A decision flow diagram for the issues addressed in Appendix B is provided in Fig. 3 below.


EVAP
Get Propagation Channel Data
for all paths
for bottom bounce and ducted paths only
Deterministic path parameter: travel time, amplitude, pathID, etc
NFinePaths (≤ MaxFinePaths)
Source Virtual Depth
Receiver Virtual Depth
FineDelay Correlation Distance
Amplitude, FineDelay, Absorption, PatchWidth, PatchLength, PatchOffset
Fine Path #1
Amplitude, FineDelay, Absorption, PatchWidth, PatchLength, PatchOffset
Fine Path #2
NFinePaths
for each fine path
Is PatchWidth = 0?
yes
Use given Amplitude and FineDelay for fine path calculation
combine with deterministic data
no
Have fine paths been computed previously?
yes
Compute source and receiver displacements since last fine path calc, sum and divide by FineDelay Correlation Distance
no

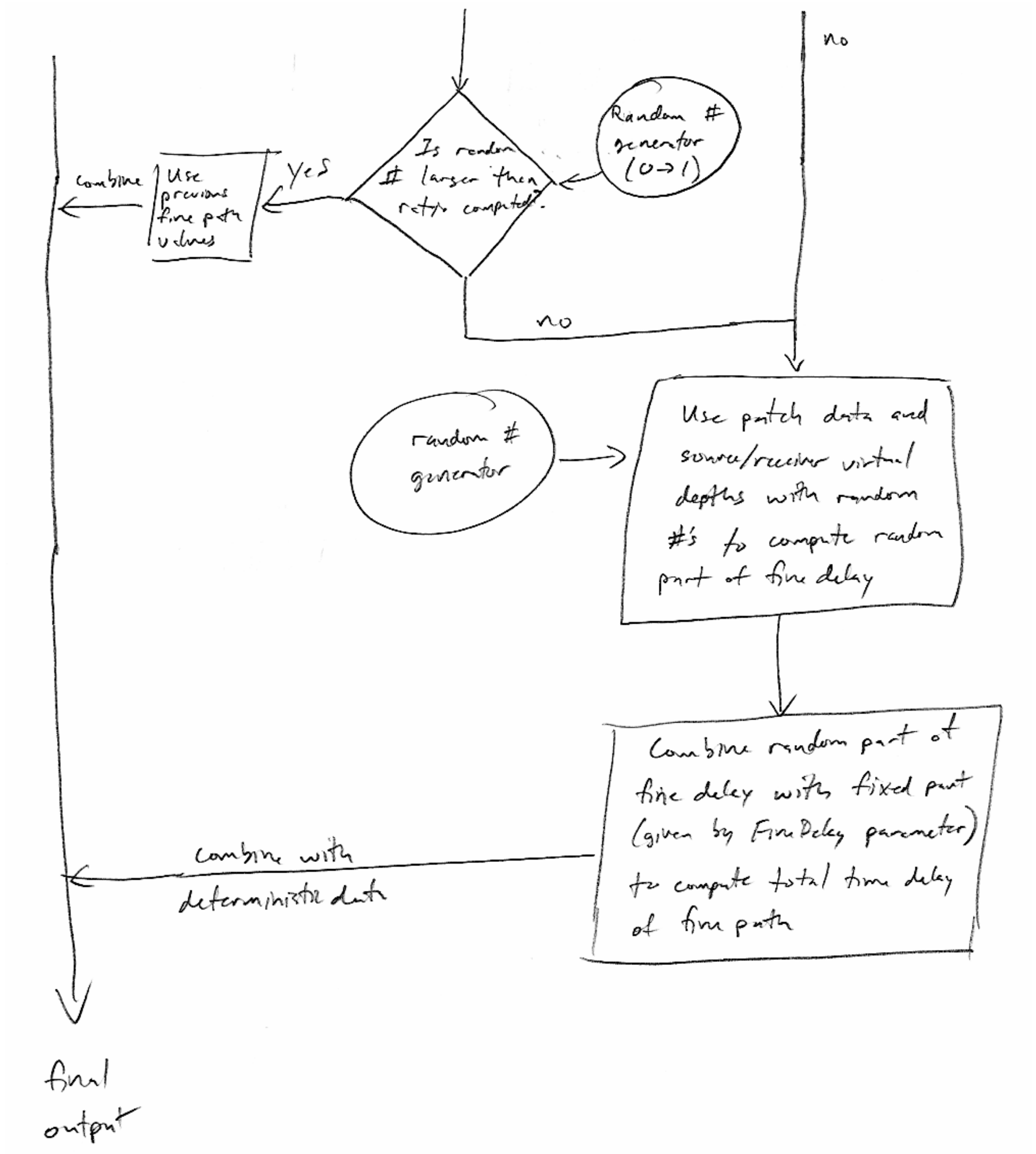

no
Random # generator (0→1)
Is random # larger than ratio computed?
yes
Use previous fine path values
combine
no
random # generator
Use patch data and source/receiver virtual depths with random #'s to compute random part of fine delay
Combine random part of fine delay with fixed part (given by FineDelay parameter) to compute total time delay of fine path
combine with deterministic data
final output

Figure 3. Decision flow for fine path travel time fluctuations as described in EVA Manual Appendix B.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 8725 John J. Kingman Rd., STE 0944 Ft. Belvoir, VA 22060-6218	2
2. Dudley Knox Library, Code 52 Naval Postgraduate School Monterey, CA 93953	2
3. Research Office, Code 09 Naval Postgraduate School Monterey, CA 93953	1
4. Cathy Matthews NAVAIR Code 4613 12350 Research Parkway Orlando, FL 32826	2
5. Prof. V. Berzins, CS/Be Naval Postgraduate School Monterey, CA 93953	1
6. Prof. K. Smith, PH/Sk Naval Postgraduate School Monterey, CA 93953	1